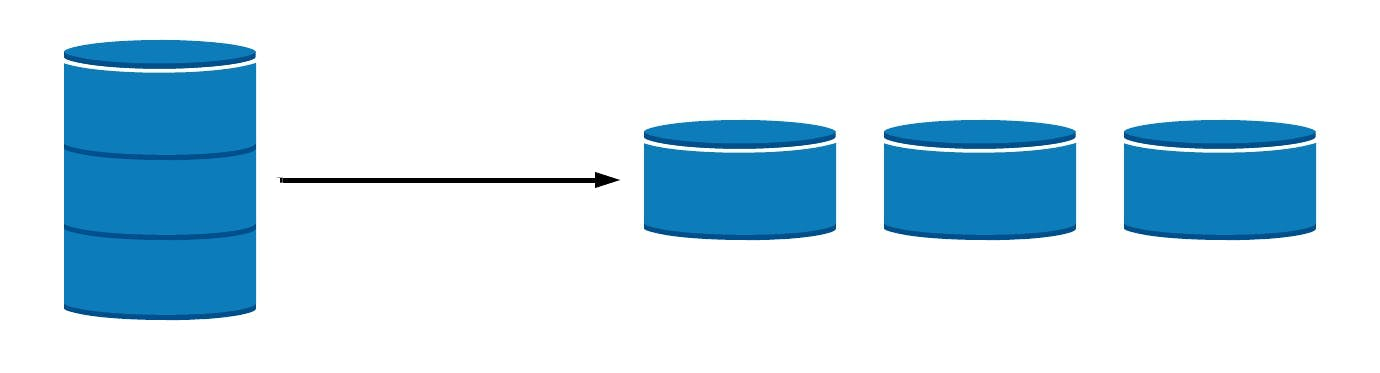
**Sharding** is a **method for distributing a single dataset across multiple databases**, which **can then be stored on multiple machines**. This **allows for larger datasets** to be **split into smaller chunks** and **stored in multiple data nodes**, **increasing the total storage capacity of the system**.

**Sharding** is a **form of scaling known as horizontal scaling** or **scale-out**, as **additional nodes are brought on to share the load**. **Horizontal scaling allows for near-limitless scalability to handle big data and intense workloads**. In contrast, vertical scaling refers to increasing the power of a single machine or single server through a more powerful CPU, increased RAM, or increased storage capacity.



**Advantages of Sharding**

* **Increased Read/Write Throughput** — **By distributing the dataset across multiple shards**, both **read and write operation capacity is increased** as long as **read and write operations are confined to a single shard**.
* **Increased Storage Capacity** — Similarly, **by increasing the number of shards**, we can **also increase overall total storage capacity**, **allowing near-infinite scalability**.
* **High Availability** — Finally, **shards provide high availability** in two ways. First, since **each shard is a replica set**, **every piece of data is replicated**. Second, **even if an entire shard becomes unavailable since the data is distributed**, the **database as a whole still remains partially functional**, with part of the schema on different shards.

**Disadvantages of Sharding**

* **Query Overhead** — **Each sharded database** must have a **separate machine or service** which **understands how to route a querying operation to the appropriate shard**. This **introduces additional latency on every operation**. Furthermore, **if the data required for the query is horizontally partitioned across multiple shards**, the **router must then query each shard and merge the result together**. This can make an otherwise simple **operation quite expensive** and **slow down response times**.
* **Complexity of Administration** — With a single unsharded database, only the database server itself requires upkeep and maintenance. **With every sharded database**, **on top of managing the shards themselves, there are additional service nodes to maintain**. Plus, in **cases where replication is being used**, **any data updates must be mirrored across each replicated node**. Overall, a sharded database is a more complex system that requires more administration.
* **Increased Infrastructure Costs** — **Sharding** **by its nature** **requires additional machines and compute power** over a single database server. While this allows your database to grow beyond the limits of a single machine, each **additional shard comes with higher costs**. The cost of a distributed database system, especially if it is missing the proper optimization, can be significant.

**How Does Sharding Work?**

In order to shard a database, we must answer several fundamental questions. The answers will determine our implementation.

**First, how will the data be distributed across shards?**

* This is the fundamental question behind any sharded database. The **answer to this question will have effects on both performance and maintenance**. More detail on this can be found in the “Sharding Architectures and Types” section.

**Second, what types of queries will be routed across shards?**

* **If the workload is primarily read operations**, **replicating data will be highly effective at increasing performance**, and you **may not need sharding** at all. In contrast, **a mixed read-write workload** or even a **primarily write-based workload** will **require a different architecture**.

**Finally, how will these shards be maintained?**

* Once you have sharded a database, over time, **data will need to be redistributed among the various shards**, and **new shards may need to be created**. Depending on the distribution of data, this can be an **expensive process** and **should be considered ahead of time**.

With these questions in mind, let’s consider some sharding architectures.

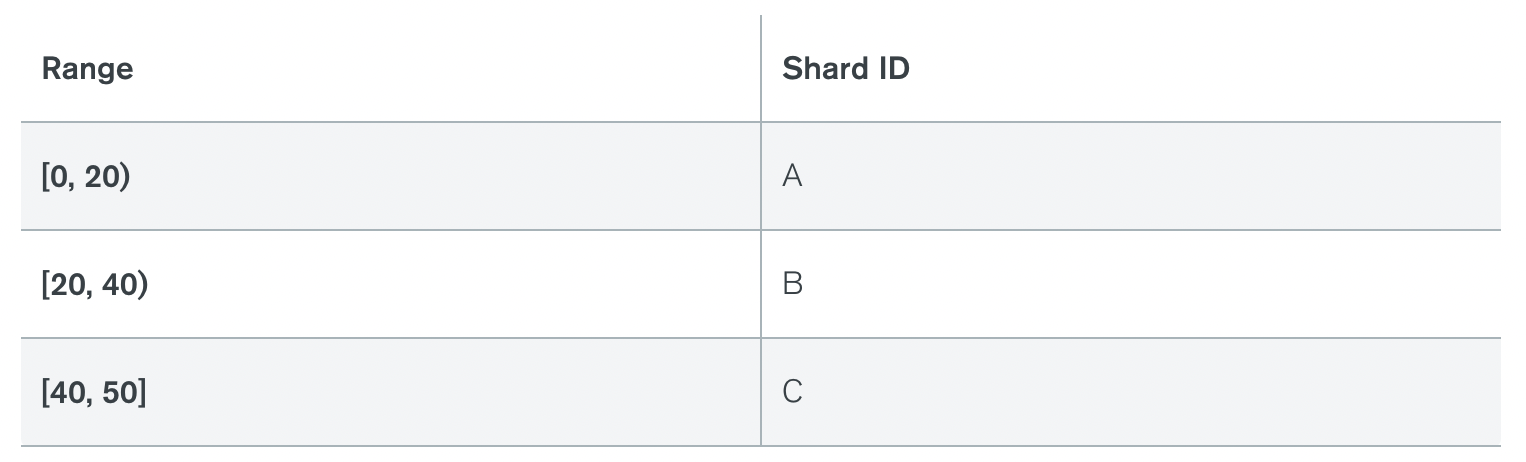
## 

## **Sharding Architectures and Types**

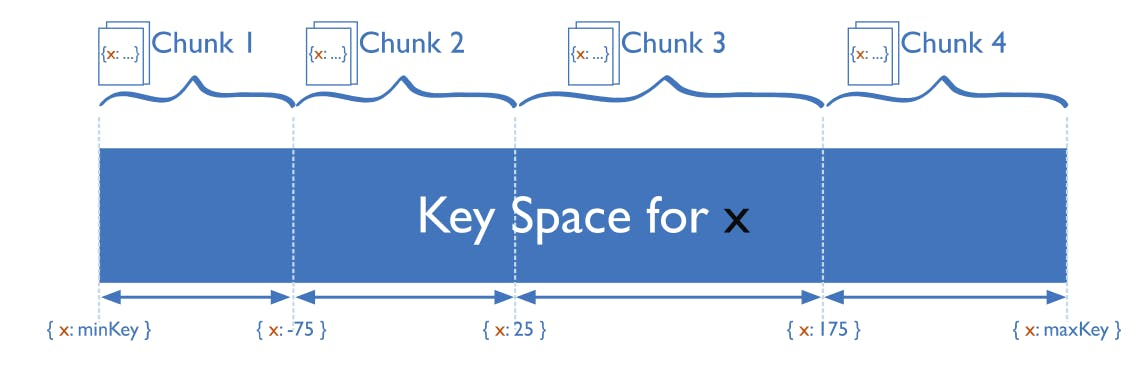
### **Ranged/Dynamic Sharding**

[**Ranged sharding**](https://docs.mongodb.com/manual/core/ranged-sharding/), **or dynamic sharding**, **takes a field on the record as an input** and, **based on a predefined range**, **allocates that record to the appropriate shard**. **Ranged sharding requires** there to be **a lookup table or service available for all queries or writes**.

For example, consider a set of data with IDs that range from 0-50. A simple lookup table might look like the following:



The **field on which the range is based** is also **known as** the [**shard key**](https://docs.mongodb.com/manual/core/sharding-shard-key/). Naturally, **the choice of shard key**, **as well as the ranges**, **are critical in making range-based sharding effective**. A **poor choice of shard key will lead to unbalanced shards**, which leads to decreased performance. An **effective shard key will allow for queries to be targeted to a minimum number of shards**. In our example above, if we query for all records with IDs 10-30, then only shards A and B will need to be queried.



**Two key attributes of an effective shard key** are **high cardinality** and **well-distributed frequency**. **Cardinality refers to the number of possible values of that key**. If a shard key only has three possible values, then there can only be a maximum of three shards. **Frequency refers to the distribution of the data along with the possible values**. If 95% of records occur with a single shard key value, then due to this hotspot, 95% of the records will be allocated to a single shard. Consider both of these attributes when selecting a shard key.

**Range-based sharding** is an **easy-to-understand method of horizontal partitioning**, but its **effectiveness** **depends heavily** on the **availability of a suitable shard key** and the **selection of appropriate ranges**. Additionally, the lookup service can become a bottleneck, although the amount of data is small enough that this typically is not an issue.

### **Algorithmic/Hashed Sharding**

**Algorithmic sharding or** [**hashed sharding**](https://docs.mongodb.com/manual/core/hashed-sharding/) **takes a record as an input** and **applies a hash function or algorithm** to it **which generates an output or hash value**. **This output is then used to allocate each record to the appropriate shard**.

The function can take any subset of values on the record as inputs. Perhaps the simplest example of a hash function is to use the modulus operator with the number of shards, as follows:

**Hash Value = ID % Number of Shards**

This is **similar to range-based sharding**—**a set of fields determines the allocation of the record to a given shard**. **Hashing the inputs allows more even distribution across shards** **even when there is not a suitable shard key**, and **no lookup table needs to be maintained**.

However, there are a few drawbacks.

First**, query operations for multiple records are more likely to get distributed across multiple shards**. Whereas ranged sharding reflects the natural structure of the data across shards, hashed sharding typically disregards the meaning of the data. This is reflected in increased broadcast operation occurrence.

Second, **resharding can be expensive**. **Any update to the number of shards likely requires rebalancing all shards to move around records**. It will be difficult to do this while avoiding a system outage.

### **Entity-/Relationship-Based Sharding**

**Entity-based sharding** **keeps related data together on a single physical shard**. In a relational database (such as PostgreSQL, MySQL, or SQL Server), related data is often spread across several different tables.

For instance, consider the case of a shopping database with Users and Payment Methods. Each user has a set of payment methods that are tied tightly with that user. As such, **keeping related data together on the same shard** **can** **reduce the need for broadcast operations, increasing performance**.

### **Geography-Based Sharding**

**Geography-based sharding**, or **geosharding**, also **keeps related data together on a single shard**, but in this case, **the data is related by geography**. This is **essentially ranged sharding** where the **shard key contains geographic information and the shards themselves are geo-located**.

For example, consider a dataset where each record contains a “country” field. In this case, we can both **increase overall performance and decrease system latency** **by creating a shard for each country or region** **and** **storing the appropriate data on that shard**.